Document number: N3985
Date: 2014-05-22
Project: Programming Language C++, SG1
Reply-to: Oliver Kowalke (oliver dot kowalke at gmail dot com)
Nat Goodspeed (nat at lindenlab dot com)

A proposal to add coroutines to the C++ standard library (Revision 1)

Revision History This document supersedes N3708. A new kind of coroutines - `std::symmetric_coroutine<T>` - is introduced and additional examples (like recursive SAX parsing) are added.
A section explains the benfits of using coroutines in the context of event-based asynchronous model.

Introduction

This proposal suggests adding two first-class continuations to the C++ standard library:
`std::asymmetric_coroutine<T>` and `std::symmetric_coroutine<T>` .

In computer science routines are defined as a sequence of operations. The execution of routines forms a parent-child relationship and the child terminates always before the parent. Coroutines (the term was introduced by Melvin Conway[1]) are a generalization of routines (Donald Knuth[2]). The principal difference between coroutines and routines is that a coroutine enables explicit suspend and resume of its progress via additional operations by preserving execution state and thus provides an **enhanced control flow** (maintaining the execution context).

characteristics: Characteristics[3] of a coroutine are:

- values of local data persist between successive calls (context switches)
- execution is suspended as control leaves coroutine and resumed at certain time later
- symmetric or asymmetric control-transfer mechanism; see below
- first-class object (can be passed as argument, returned by procedures, stored in a data structure to be used later or freely manipulated by the developer)
- stackful or stackless

Several programming languages adopted particular features (C# yield, Python generators, ...).

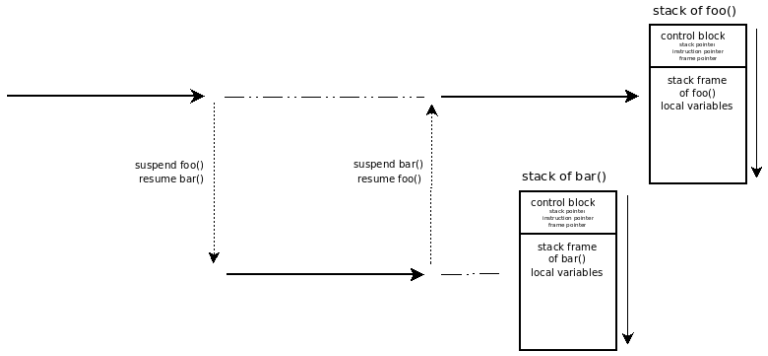
BCPL	Erlang	Go	Lua	PHP	Ruby
C#	F#	Haskell	Modula-2	Prolog	Sather
D	Factor	Icon	Perl	Python	Scheme

Table 1: some programming languages with native support of coroutines[17]

Coroutines are useful in simulation, artificial intelligence, concurrent programming, text processing and data manipulation,[3] supporting the implementation of components such as cooperative tasks (fibers), iterators, generators, infinite lists, pipes etc.

execution-transfer mechanism: Two categories of coroutines exist: symmetric and asymmetric coroutines.

An asymmetric coroutine knows its invoker, using a special operation to implicitly yield control specifically to its invoker. By contrast, all symmetric coroutines are equivalent; one symmetric coroutine may pass control to any other symmetric coroutine. Because of this, a symmetric coroutine *must* specify the coroutine to which it intends to yield control.



stackfulness: In contrast to a stackless coroutine a stackful coroutine can be suspended from within a nested stackframe. The execution resumes at the exact same point in the code where it was suspended before.
With a stackless coroutine, only the top-level routine may be suspended. Any routine called by that top-level routine may not itself suspend. This prohibits providing suspend/resume operations in routines within a general-purpose library.

first-class continuation: A first-class continuation can be passed as an argument, returned by a function and stored in a data structure to be used later.
In some implementations (for instance C# *yield*) the continuation can not be directly accessed or directly manipulated.

Without stackfulness and first-class semantics some useful execution control flows cannot be supported (for instance cooperative multitasking or checkpointing).

What coroutines actually do: Coroutines are generalized routines.

A routine has a parent-child relationship to its subroutines.
The routine processes (pushes to stack or stores in registers) the arguments which have to be passed as parameters to the subroutine as it is defined in the calling convention[11] of the underlying ABI[10]. A *branch-and-link* instruction transfers execution control to the code of the subroutine.
When the subroutine is entered the *prolog* creates a new stack frame (adjusting the stack-pointer and/or frame-pointer), preserves some non-volatile general purpose registers (as defined by the calling convention[11]) and return address. Space for local variables is allocated by modifying the stack-pointer.
When a subroutine finishes, it runs the *epilog* which undoes the steps from the *prolog*, e.g. it restores preserved registers, removes the stack-frame (stack-pointer is restored to address before the subroutine was entered), and branches to the instruction at the return address.
A return value might be returned in a register defined by the calling convention[11].

When a coroutine switches execution context it executes the same as ordinary routines: saving and restoring some CPU registers. The main difference is that each coroutine owns its own stack, that is, when a coroutine is suspended its stackframe is not removed. This fact is the reason why a coroutine allows you to resume from the suspend point.
A coroutine contains a *control-block* which is used as a storage for the stack-pointer, instruction-pointer and some general purpose registers.
Coroutines manipulate those registers directly (by calling *suspend*/*resume*).
Appendix A. *jump*-operation for SYSV ABI on x86_64 shows for x86_64/SYSV ABI how a *jump*-operation could be implemented.
An ordinary routine can be regarded as a degenerate coroutine that does not suspend and runs straight to its end (finishes execution).

In fact entering a coroutine is equivalent to entering an ordinary routine, but it also supports suspend and resume.

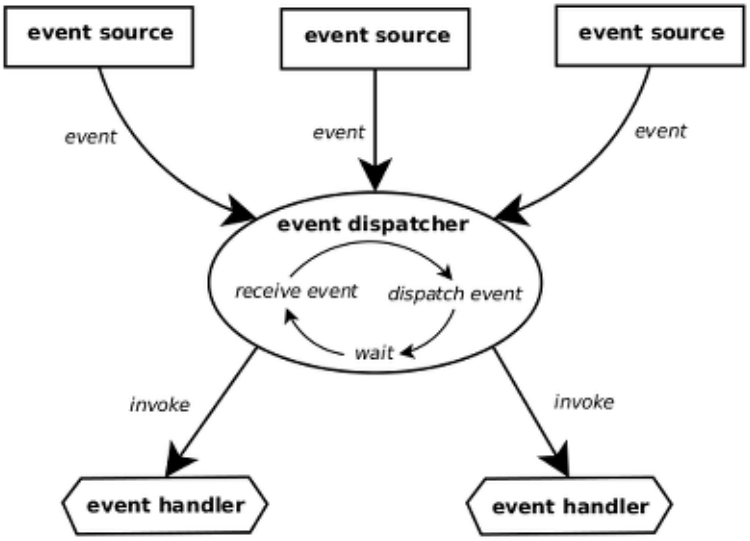
Motivation

This proposal refers to boost.coroutine[8] as reference implementation - providing a test suite and examples (some are described in this section).

In order to support a broad range of execution control behaviour the coroutine types of `std::symmetric_coroutine<T>` and `std::asymmetric_coroutine<T>` can be used to *escape-and-reenter loops*, to *escape-and-reenter recursive computations* and for *cooperative multitasking* helping to solve problems in a much simpler and more elegant way than with only a single flow of control.

event-driven model

The event-driven model is a programming paradigm where the flow of a program is determined by events. The events are generated by multiple independent sources and an event-dispatcher, waiting on all external sources, triggers callback functions (event-handlers) whenever one of those events is detected (event-loop). The application is divided into event selection (detection) and event handling.



The resulting applications are highly scalable, flexible, have high responsiveness and the components are loosely coupled. This makes the event-driven model suitable for user interface applications, rule-based productions systems or applications dealing with asynchronous I/O (for instance network servers).

event-based asynchronous paradigm

A classic synchronous console program issues an I/O request (e.g. for user input or filesystem data) and blocks until the request is complete.
In contrast, an asynchronous I/O function initiates the physical operation but immediately returns to its caller, even though the operation is not yet complete. A program written to leverage this functionality does not block: it can proceed with other work (including other I/O requests in parallel) while the original operation is still pending. When the operation completes, the program is notified. Because asynchronous applications spend less overall time waiting for operations, they can outperform synchronous programs.
Events are one of the paradigms for asynchronous execution, but not all asynchronous systems use events. Although asynchronous programming can be done using threads, they come with their own costs:

- hard to program (traps for the unwary)
- memory requirements are high
- large overhead with creation and maintenance of thread state
- expensive context switching between threads

The event-based asynchronous model avoids those issues:

- simpler because of the single stream of instructions
- much less expensive context switches

The downside of this paradigm consists in a sub-optimal program structure. An event-driven program is required to split its code into multiple small callback functions, i.e. the code is organized in a sequence of small steps that execute intermittently. An algorithm that would usually be expressed as a hierarchy of functions and loops must be transformed into callbacks. The complete state has to be stored into a data structure while the control flow returns to the event-loop. As a consequence, event-driven applications are often tedious and confusing to write. Each callback introduces a new scope, error callback etc. The sequential nature of the algorithm is split into multiple callstacks, making the application hard to debug. Exception handlers are restricted to local handlers: it is impossible to wrap a sequence of events into a single try-catch block. The use of local variables, while/for loops, recursions etc. together with the event-loop is not possible. The code becomes less expressive.

In the past, code using asio's *asynchronous-operations* was convoluted by callback functions.

```
class session{
public:
    session(boost::asio::io_service& io_service) :
          socket_(io_service) // construct a TCP-socket from io_service
    {}

    tcp::socket& socket(){
        return socket_;
    }

    void start(){
        // initiate asynchronous read; handle_read() is callback-function
        socket_.async_read_some(boost::asio::buffer(data_,max_length),
            boost::bind(&session::handle_read,this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred));
    }

private:
    void handle_read(const boost::system::error_code& error,
                     size_t bytes_transferred){
        if (!error)
            // initiate asynchronous write; handle_write() is callback-function
            boost::asio::async_write(socket_,
                boost::asio::buffer(data_,bytes_transferred),
                boost::bind(&session::handle_write,this,
                    boost::asio::placeholders::error));
        else
            delete this;
    }

    void handle_write(const boost::system::error_code& error){
        if (!error)
            // initiate asynchronous read; handle_read() is callback-function
            socket_.async_read_some(boost::asio::buffer(data_,max_length),
                boost::bind(&session::handle_read,this,
                    boost::asio::placeholders::error,
                    boost::asio::placeholders::bytes_transferred));
        else
            delete this;
    }

    boost::asio::ip::tcp::socket socket_;
    enum { max_length=1024 };
    char data_[max_length];
};
```

In this example, a simple echo server, the logic is split into three member functions - local state (such as data buffer) is moved to member variables.

boost.asio[6] provides with its new *asynchronous-result* feature a new framework combining event-driven model and

coroutines, hiding the complexity of event-driven programming and permitting the style of classic sequential code. The application is not required to pass callback functions to asynchronous operations and local state is kept as local variables. Therefore the code is much easier to read and understand. Proposal 'N3964: Library Foundations for Asynchronous Operations'[5] describes the usage of coroutines in the context of asynchronous operations.
yield_context[18] internally uses boost.coroutine[8]:

```
void session(boost::asio::io_service& io_service,
             boost::asio::ip::tcp::socket& socket){

    try{
        for(;;){
            // local data-buffer
            char data[max_length];

            boost::system::error_code ec;

            // read asynchronous data from socket
            // execution context will be suspended until
            // some bytes are read from socket
            std::size_t length=socket.async_read_some(
                    boost::asio::buffer(data),
                    boost::asio::yield[ec]);
            if (ec==boost::asio::error::eof)
                break; //connection closed cleanly by peer
            else if(ec)
                throw boost::system::system_error(ec); //some other error

            // write some bytes asynchronously
            boost::asio::async_write(
                    socket,
                    boost::asio::buffer(data,length),
                    boost::asio::yield[ec]);
            if (ec==boost::asio::error::eof)
                break; //connection closed cleanly by peer
            else if(ec)
                throw boost::system::system_error(ec); //some other error
        }
    } catch(std::exception const& e){
        std::cerr<<"Exception: "<<e.what()<<"\n";
    }
}
```

In contrast to the previous example this one gives the impression of sequential code and local data while using asynchronous operations `async_read` and `async_write` . The algorithm is implemented in one function and error handling is done by one try-catch block.

recursive SAX parsing

To someone who knows SAX, the phrase "recursive SAX parsing" might sound nonsensical. You get callbacks from SAX; you have to manage the element stack yourself. If you want recursive XML processing, you must first read the entire DOM into memory, then walk the tree.
But coroutines let you invert the flow of control so you can ask for SAX events. Once you can do that, you can process them recursively.

```
// Represent a subset of interesting SAX events
struct BaseEvent{
   BaseEvent(const BaseEvent&)=delete;
   BaseEvent& operator=(const BaseEvent&)=delete;
};

// End of document or element
struct CloseEvent: public BaseEvent{
   // CloseEvent binds (without copying) the TagType reference.
   CloseEvent(const xml::sax::Parser::TagType& name):
```

```
        mName(name)
    {}

    const xml::sax::Parser::TagType& mName;
};

// Start of document or element
struct OpenEvent: public CloseEvent{
    // In addition to CloseEvent's TagType, OpenEvent binds AttributeIterator.
    OpenEvent(const xml::sax::Parser::TagType& name,
              xml::sax::AttributeIterator& attrs):
        CloseEvent(name),
        mAttrs(attrs)
    {}

    xml::sax::AttributeIterator& mAttrs;
};

// text within an element
struct TextEvent: public BaseEvent{
    // TextEvent binds the CharIterator.
    TextEvent(xml::sax::CharIterator& text):
        mText(text)
    {}

    xml::sax::CharIterator& mText;
};

// The parsing coroutine instantiates BaseEvent subclass instances and
// successively shows them to the main program. It passes a reference so we
// don't slice the BaseEvent subclass.
typedef std::asymmetric_coroutine<const BaseEvent&> coro_t;

void parser(coro_t::push_type& sink,std::istream& in){
    xml::sax::Parser xparser;
    // startDocument() will send OpenEvent
    xparser.startDocument([&sink](const xml::sax::Parser::TagType& name,
                                  xml::sax::AttributeIterator& attrs)
                          {
                              sink(OpenEvent(name,attrs));
                          });
    // startTag() will likewise send OpenEvent
    xparser.startTag([&sink](const xml::sax::Parser::TagType& name,
                             xml::sax::AttributeIterator& attrs)
                     {
                         sink(OpenEvent(name,attrs));
                     });
    // endTag() will send CloseEvent
    xparser.endTag([&sink](const xml::sax::Parser::TagType& name)
                   {
                       sink(CloseEvent(name));
                   });
    // endDocument() will likewise send CloseEvent
    xparser.endDocument([&sink](const xml::sax::Parser::TagType& name)
                        {
                            sink(CloseEvent(name));
                        });
    // characters() will send TextEvent
    xparser.characters([&sink](xml::sax::CharIterator& text)
                       {
                           sink(TextEvent(text));
                       });
```

```
    try
    {
        // parse the document, firing all the above
        xparser.parse(in);
    }
    catch (xml::Exception e)
    {
        // xml::sax::Parser throws xml::Exception. Helpfully translate the
        // name and provide it as the what() string.
        throw std::runtime_error(exception_name(e));
    }
}

// Recursively traverse the incoming XML document on the fly, pulling
// BaseEvent& references from 'events'.
// 'indent' illustrates the level of recursion.
// Each time we're called, we've just retrieved an OpenEvent from 'events';
// accept that as a param.
// Return the CloseEvent that ends this element.
const CloseEvent& process(coro_t::pull_type& events,const OpenEvent& context,
                          const std::string& indent=""){
    // Capture OpenEvent's tag name: as soon as we advance the parser, the
    // TagType& reference bound in this OpenEvent will be invalidated.
    xml::sax::Parser::TagType tagName = context.mName;
    // Since the OpenEvent is still the current value from 'events', pass
    // control back to 'events' until the next event. Of course, each time we
    // come back we must check for the end of the results stream.
    while(events()){
        // Another event is pending; retrieve it.
        const BaseEvent& event=events.get();
        const OpenEvent* oe;
        const CloseEvent* ce;
        const TextEvent* te;
        if((oe=dynamic_cast<const OpenEvent*>(&event))){
            // When we see OpenEvent, recursively process it.
            process(events,*oe,indent+"␣␣␣␣");
        }
        else if((ce=dynamic_cast<const CloseEvent*>(&event))){
            // When we see CloseEvent, validate its tag name and then return
            // it. (This assert is really a check on xml::sax::Parser, since
            // it already validates matching open/close tags.)
            assert(ce->mName == tagName);
            return *ce;
        }
        else if((te=dynamic_cast<const TextEvent*>(&event))){
            // When we see TextEvent, just report its text, along with
            // indentation indicating recursion level.
            std::cout<<indent<<"text:␣'"<<te->mText.getText()<<"'\n";
        }
    }
}

// pretend we have an XML file of arbitrary size
std::istringstream in(doc);
try
{
    coro_t::pull_type events(std::bind(parser,_1,std::ref(in)));
    // We fully expect at least ONE event.
    assert(events);
    // This dynamic_cast<&> is itself an assertion that the first event is an
    // OpenEvent.
    const OpenEvent& context=dynamic_cast<const OpenEvent&>(events.get());
```

```
    process(events, context);
}
catch (std::exception& e)
{
    std::cout << "Parsing␣error:␣" << e.what() << '\n';
}
```

This problem does not map at all well to communicating between independent threads. It makes no sense for either side to proceed independently of the other. You want them to pass control back and forth.
The solution involves a small polymorphic class event hierarchy, to which we're passing references. The actual instances are temporaries on the coroutine's stack; the coroutine passes each reference in turn to the main logic. Copying them as base-class values would slice them.
If we were trying to let the SAX parser proceed independently of the consuming logic, one could imagine allocating event-subclass instances on the heap, passing them along on a thread-safe queue of pointers. But that doesn't work either, because these event classes bind references passed by the SAX parser. The moment the parser moves on, those references become invalid.
Instead of binding a `TagType&` reference, we could store a copy of the `TagType` in `CloseEvent`. But that doesn't solve the whole problem. For attributes, we get an `AttributeIterator&`; for text we get a `CharIterator&`. Storing a copy of those iterators is pointless: once the parser moves on, those iterators are invalidated. You must process the attribute iterator (or character iterator) during the SAX callback for that event.
Naturally we could retrieve and store a copy of every attribute and its value; we could store a copy of every chunk of text. That would effectively be all the text in the document – a heavy price to pay, if the reason we're using SAX is concern about fitting the entire DOM into memory.
There's yet another advantage to using coroutines. This SAX parser throws an exception when parsing fails. With a coroutine implementation, you need only wrap the calling code in try/catch.
With communicating threads, you would have to arrange to catch the exception and pass along the exception pointer on the same queue you're using to deliver the other events. You would then have to rethrow the exception to unwind the recursive document processing.
The coroutine solution maps very naturally to the problem space.

'same fringe' problem

The advantages of stackful coroutines can be seen particularly clearly with the use of a recursive function, such as traversal of trees.
If traversing two different trees in the same deterministic order produces the same list of leaf nodes, then both trees have the same fringe even if the tree structure is different.

The same fringe problem could be solved using coroutines by iterating over the leaf nodes and comparing this sequence via `std::equal()`. The range of data values is generated by function `traverse()` which recursively traverses the tree and passes each node's data value to its `std::asymmetric_coroutine<T>::push_type`.
`std::asymmetric_coroutine<T>::push_type` suspends the recursive computation and transfers the data value to the main execution context.
`std::asymmetric_coroutine<T>::pull_type::iterator`, created from `std::asymmetric_coroutine<T>::pull_type`, steps over those data values and delivers them to `std::equal()` for comparison. Each increment of `std::asymmetric_coroutine<T>::pull_type::iterator` resumes `traverse()`. Upon return from `iterator::operator++()`, either a new data value is available, or tree traversal is finished (iterator is invalidated).
In effect, the coroutine iterator presents a flattened view of the recursive data structure.

```
struct node{
    typedef std::shared_ptr<node> ptr_t;

    // Each tree node has an optional left subtree,
    // an optional right subtree and a value of its own.
    // The value is considered to be between the left
    // subtree and the right.
    ptr_t       left,right;
    std::string value;

    // construct leaf
    node(const std::string& v):
        left(),right(),value(v)
    {}
    // construct nonleaf
```

```
    node(ptr_t l,const std::string& v,ptr_t r):
        left(l),right(r),value(v)
    {}

    static ptr_t create(const std::string& v){
        return ptr_t(new node(v));
    }

    static ptr_t create(ptr_t l,const std::string& v,ptr_t r){
        return ptr_t(new node(l,v,r));
    }
};

node::ptr_t create_left_tree_from(const std::string& root){
    /* --------
         root
         / \
        b   e
       / \
      a   c
     -------- */
    return node::create(
            node::create(
                node::create("a"),
                "b",
                node::create("c")),
            root,
            node::create("e"));
}

node::ptr_t create_right_tree_from(const std::string& root){
    /* --------
         root
         / \
        a   d
           / \
          c   e
       -------- */
    return node::create(
            node::create("a"),
            root,
            node::create(
                node::create("c"),
                "d",
                node::create("e")));
}

// recursively walk the tree, delivering values in order
void traverse(node::ptr_t n,
              std::asymmetric_coroutine<std::string>::push_type& out){
    if(n->left) traverse(n->left,out);
    out(n->value);
    if(n->right) traverse(n->right,out);
}

// evaluation
{
    node::ptr_t left_d(create_left_tree_from("d"));
    std::asymmetric_coroutine<std::string>::pull_type left_d_reader(
        [&](std::asymmetric_coroutine<std::string>::push_type& out){
            traverse(left_d,out);
        });
```

```
    node::ptr_t right_b(create_right_tree_from("b"));
    std::asymmetric_coroutine<std::string>::pull_type right_b_reader(
        [&](std::asymmetric_coroutine<std::string>::push_type& out){
            traverse(right_b,out);
        });

    std::cout << "left tree from d == right tree from b? "
              << std::boolalpha
              << std::equal(std::begin(left_d_reader),
                            std::end(left_d_reader),
                            std::begin(right_b_reader))
              << std::endl;
}
{
    node::ptr_t left_d(create_left_tree_from("d"));
    std::asymmetric_coroutine<std::string>::pull_type left_d_reader(
        [&](std::asymmetric_coroutine<std::string>::push_type& out){
            traverse(left_d,out);
        });

    node::ptr_t right_x(create_right_tree_from("x"));
    std::asymmetric_coroutine<std::string>::pull_type right_x_reader(
        [&](std::asymmetric_coroutine<std::string>::push_type& out){
            traverse(right_x,out);
        });

    std::cout << "left tree from d == right tree from x? "
              << std::boolalpha
              << std::equal(std::begin(left_d_reader),
                            std::end(left_d_reader),
                            std::begin(right_x_reader))
              << std::endl;
}
std::cout << "Done" << std::endl;

output:
left tree from d == right tree from b? true
left tree from d == right tree from x? false
Done
```

C# await

C# contains the two keywords *async* and *await*. *async* introduces a control flow that involves awaiting asynchronous operations. The compiler reorganizes the code into a continuation-passing style. *await* wraps the rest of the function after *await* into a continuation if the asynchronous operation has not yet completed.
The project await_emu[12] uses boost.coroutine[8] for a proof-of-concept demonstrating the implementation of a full emulation of *C# await* as a library extension. Because of stackful coroutines *await* is **not limited** by "one level" as in *C#*.
Evgeny Panasyuk describes the advantages of boost.coroutine[8] over *await* at Channel 9 - 'The Future of C++'[12].

```
int bar(int i){
    // await is not limited by "one level" as in C#
    auto result=await async([i]{ return reschedule(),i*100; });
    return result+i*10;
}

int foo(int i){
    cout << i << ":\tbegin" << endl;
    cout << await async([i]{ return reschedule(),i*10; }) << ":\tbody" << endl;
    cout << bar(i) << ":\tend" << endl;
    return i*1000;
```

```
}

void async_user_handler(){
    vector<future<int>> fs;

    // instead of ‘async‘ at function signature, ‘asynchronous‘ should be
    // used at the call place:
    for(auto i=0;i!=5;++i)
        fs.push_back(asynchronous([i]{ return foo(i+1); }));

    for(auto&& f:fs)
        cout << await f << ":\tafter␣end" << endl;
}
```

Impact on the Standard

This proposal is a library extension. It does not require changes to any standard classes, functions or headers. It can be implemented in C++03 and C++11 and requires no core language changes.

Design Decisions

Proposed Design

The design suggests two kinds of coroutines - `std::asymmetric_coroutine<T>` and `std::symmetric_coroutine<T>`. Symmetric coroutines usually occur in the context of concurrent programming in order to represent independent units of execution. Implementations that produce sequences of values typically use asymmetric coroutines.[3]

std::asymmetric_coroutine<>::pull_type: provides an asymmetric execution-transfer mechanism. This type transfers data from another execution context (== pulled-from).
The class has only one template parameter defining the transferred parameter type.
The constructor of `std::asymmetric_coroutine<T>::pull_type` takes a function (*coroutine-function*) accepting a reference to a `std::asymmetric_coroutine<T>::push_type` as argument.
Instantiating a `std::asymmetric_coroutine<T>::pull_type` passes the control of execution to *coroutine-function* and a complementary `std::asymmetric_coroutine<T>::push_type` is synthesized by the library and passed as reference to *coroutine-function*.

This kind of coroutine provides `std::asymmetric_coroutine<T>::pull_type::operator()()` - this method only switches context; it transfers no data.

`std::asymmetric_coroutine<T>::pull_type` provides input iterators
(`std::asymmetric_coroutine<T>::pull_type::iterator`) and `std::begin()` / `std::end()` are overloaded. The increment-operation switches the context and transfers data.

```
std::asymmetric_coroutine<int>::pull_type source(
    [&](std::asymmetric_coroutine<int>::push_type& sink){
        int first=1,second=1;
        sink(first);
        sink(second);
        for(int i=0;i<8;++i){
            int third=first+second;
            first=second;
            second=third;
            sink(third);
        }
    });

for(auto i:source)
    std::cout << i <<  "␣";

std::cout << "\nDone" << std::endl;

output:
```

```
1 1 2 3 5 8 13 21 34 55
Done
```

In this example a `std::asymmetric_coroutine<T>::pull_type` is created in the main execution context taking a lambda function (== *coroutine-function*) which calculates Fibonacci numbers in a simple *for*-loop).
The *coroutine-function* is executed in a newly created execution context which is managed by the instance of `std::asymmetric_coroutine<T>::pull_type` .
A `std::asymmetric_coroutine<T>::push_type` is automatically generated by the library and passed as reference to the lambda function. Each time the lambda function calls `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` with another Fibonacci number, `std::asymmetric_coroutine<T>::push_type` transfers it back to the main execution context. The local state of *coroutine-function* is preserved and will be restored upon transferring execution control back to *coroutine-function* to calculate the next Fibonacci number.
Because `std::asymmetric_coroutine<T>::pull_type` provides input iterators and `std::begin()` / `std::end()` are overloaded, a *range-based for*-loop can be used to iterate over the generated Fibonacci numbers.

std::asymmetric_coroutine<>::push_type: provides an asymmetric execution-transfer mechanism. This type transfers data to the other execution context (== pushed-to).
The class has only one template parameter defining the transferred parameter type.
The constructor of `std::asymmetric_coroutine<T>::push_type` takes a function (*coroutine-function*) accepting a reference to a `std::asymmetric_coroutine<T>::pull_type` as argument. In contrast to `std::asymmetric_coroutine<T>::pull_type` , instantiating a `std::asymmetric_coroutine<T>::push_type` does not pass the control of execution to *coroutine-function* - instead the first call of `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` synthesizes a complementary `std::asymmetric_coroutine<T>::pull_type` and passes it as reference to *coroutine-function*.

The `std::asymmetric_coroutine<T>::push_type` interface does not contain a `get()` -function: you can not retrieve values from another execution context with this kind of coroutine.

`std::asymmetric_coroutine<T>::push_type` provides output iterators
(`std::asymmetric_coroutine<T>::push_type::iterator`) and `std::begin()` / `std::end()` are overloaded. The increment-operation switches the context and transfers data.

```
struct FinalEOL{
    ~FinalEOL(){
        std::cout << std::endl;
    }
};

const int num=5, width=15;
std::asymmetric_coroutine<std::string>::push_type writer(
    [&](std::asymmetric_coroutine<std::string>::pull_type& in){
        // finish the last line when we leave by whatever means
        FinalEOL eol;
        // pull values from upstream, lay them out 'num' to a line
        for (;;){
            for(int i=0;i<num;++i){
                // when we exhaust the input, stop
                if(!in) return;
                std::cout << std::setw(width) << in.get();
                // now that we've handled this item, advance to next
                in();
            }
            // after 'num' items, line break
            std::cout << std::endl;
        }
    });

std::vector<std::string> words{
    "peas", "porridge", "hot", "peas",
    "porridge", "cold", "peas", "porridge",
    "in", "the", "pot", "nine",
    "days", "old" };
```

```
std::copy(std::begin(words),std::end(words),std::begin(writer));

output:
           peas       porridge            hot           peas       porridge
           cold           peas       porridge             in            the
            pot           nine           days            old
```

In this example a `std::asymmetric_coroutine<T>::push_type` is created in the main execution context accepting a lambda function (== *coroutine-function*) which requests strings and lays out *num* of them on each line.
This demonstrates the inversion of control permitted by coroutines. Without coroutines, a utility function to perform the same job would necessarily accept each new value as a function parameter, returning after processing that single value. That function would depend on a static state variable. A *coroutine-function*, however, can request each new value as if by calling a function – even though its caller also passes values as if by calling a function.
The *coroutine-function* is executed in a newly created execution context which is managed by the instance of `std::asymmetric_coroutine<T>::push_type` .
The main execution context passes the strings to the *coroutine-function* by calling `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` .
A `std::asymmetric_coroutine<T>::pull_type` is automatically generated by the library and passed as reference to the lambda function. The *coroutine-function* accesses the strings passed from the main execution context by calling `std::asymmetric_coroutine<T>::pull_type::get()` and lays those strings out on `std::cout` according the parameters *num* and *width*.
The local state of *coroutine-function* is preserved and will be restored after transferring execution control back to *coroutine-function*.
Because `std::asymmetric_coroutine<T>::push_type` provides output iterators and `std::begin()` / `std::end()` are overloaded, the `std::copy` algorithm can be used to iterate over the vector containing the strings and pass them one by one to the coroutine.

std::symmetric_coroutine<>::call_type: provides a symmetric execution-transfer mechanism. This type transfers data to the other execution context.
The class has only one template parameter defining the transferred parameter type.

`std::symmetric_coroutine<T>::call_type` starts a symmetric coroutine and transfers its parameter to its *coroutine-function*. The template parameter defines the transferred parameter type. The constructor of `std::symmetric_coroutine<T>::call_type` takes a function (coroutine-function) accepting a reference to a `std::symmetric_coroutine<T>::yield_type` as argument. Instantiating a `std::symmetric_coroutine<T>::call_type` does not pass the control of execution to *coroutine-function* - instead the first call of `std::symmetric_coroutine<T>::call_type::operator()()` synthesizes a `std::symmetric_coroutine<T>::yield_type` and passes it as reference to *coroutine-function*.
The `std::symmetric_coroutine<T>::call_type` interface does not contain a `get()` -function.

In contrast to `std::asymmetric_coroutine<T>` , where the relationship between caller and callee is fixed, `std::symmetric_coroutine<T>` is able to transfer execution control to any other (symmetric) coroutine. That is, a `std::symmetric_coroutine<T>` is not required to return to its direct caller.

A `std::symmetric_coroutine<T>::yield_type` is automatically generated by the library and passed as reference to the *coroutine-function*. The *coroutine-function* accesses the data passed to it by calling `std::symmetric_coroutine<T>::yield_type::get()` .
The local state of *coroutine-function* is preserved and will be restored after transferring execution control back to *coroutine-function*.

std::symmetric_coroutine<>::yield_type: provides a symmetric execution-transfer mechanism. This type transfers control to another execution-context.
`std::symmetric_coroutine<T>::yield_type::operator()()` is used to transfer data and execution control to another context by calling `std::symmetric_coroutine<T>::yield_type::operator()()` with another `std::symmetric_coroutine<T>::call_type` as first argument. Alternatively, you may transfer control back to the code that called `std::symmetric_coroutine<T>::call_type::operator()()` by calling `std::symmetric_coroutine<T>::yield_type::operator()()` without a `std::symmetric_coroutine<T>::call_type` argument.

The class has only one template parameter defining the transferred parameter type. Data transferred to the coroutine

are accessed through `std::symmetric_coroutine<T>::yield_type::get()` .

Instances of this coroutine type can be created by the library only.

```
std::vector<int> merge(const std::vector<int>& a,const std::vector<int>& b){
    std::vector<int> c;
    std::size_t idx_a=0,idx_b=0;
    std::symmetric_coroutine<void>::call_type* other_a=0,* other_b=0;

    std::symmetric_coroutine<void>::call_type coro_a(
        [&](std::symmetric_coroutine<void>::yield_type& yield){
            while(idx_a<a.size()){
                if(b[idx_b]<a[idx_a])     // element in b is less than in a
                    yield(*other_b);      // yield to coroutine coro_b
                c.push_back(a[idx_a++]); // add element to final array
            }
            // add remaining elements of array b
            while(idx_b<b.size())
                c.push_back(b[idx_b++]);
        });

    std::symmetric_coroutine<void>::call_type coro_b(
        [&](std::symmetric_coroutine<void>::yield_type& yield){
            while(idx_b<b.size()){
                if(a[idx_a]<b[idx_b])     // element in a is less than in b
                    yield(*other_a);      // yield to coroutine coro_a
                c.push_back(b[idx_b++]); // add element to final array
            }
            // add remaining elements of array a
            while(idx_a<a.size())
                c.push_back(a[idx_a++]);
        });


    other_a=&coro_a;
    other_b=&coro_b;

    coro_a(); // enter coroutine-fn of coro_a

    return c;
}

std::vector<int> a={1,5,6,10};
std::vector<int> b={2,4,7,8,9,13};
std::vector<int> c=merge(a,b);
print(a);
print(b);
print(c);

output:
    a : 1 5 6 10
    b : 2 4 7 8 9 13
    c : 1 2 4 5 6 7 8 9 10 13
```

In this example two `std::symmetric_coroutine<T>::call_type` are created in the main execution context accepting a lambda function (== coroutine-function) which merges elements of two sorted arrays into a third array. `coro_a()` enters the *coroutine-function* of `coro_a` cycling through the array and testing if the actual element in the other array is less than the element in the local one. If so, the coroutine yields to the other coroutine `coro_b` using `yield(*other_b)` . If the current element of the local array is less than the element of the other array, it is put to the third array. Because the coroutine jumps back to `coro_a()` (returning from this method) after leaving the coroutine-function, the elements of the other array will appended at the end of the third array if all element of the local array are processed.

stackful: Each instance of a coroutine has its own stack.

In contrast to stackless coroutines, stackful coroutines allow invoking the suspend operation out of arbitrary sub-stackframes, enabling *escape-and-reenter operations*.

move-only: A coroutine is moveable-only.

If it were copyable, then its stack with all the objects allocated on it would be copied too. That would force undefined behaviour if some of these objects were RAII-classes (manage a resource via RAII pattern). When the first of the coroutine copies terminates (unwinds its stack), the RAII class destructors will release their managed resources. When the second copy terminates, the same destructors will try to doubly-release the same resources, leading to undefined behavior.

clean-up: On coroutine destruction the associated stack will be unwound.

The implementer is free to deallocate the stack or cache it for future usage (for coroutines created later).

segmented stack: `std::asymmetric_coroutine<T>` and `std::symmetric_coroutine<T>` must support segmented stacks (growing on demand).

It is not always possible to accurately estimate the required stack size - in most cases too much memory is allocated (waste of virtual address-space).
At construction a coroutine starts with a default (minimal) stack size.
At this time of writing only GCC version 4.7 or higher[15] and clang version 3.4 or higher are known to support segmented stacks. With version 1.54 boost.coroutine[8] provides support for segmented stacks.

context switch: A coroutine saves and restores registers according to the underlying ABI on each context switch.

This also includes the floating point environment as required by the ABI. The implementer can omit preserving the floating point environment if he can predict that it's safe.
On POSIX systems, a coroutine context switch must not preserve signal masks for performance reasons.
A context switch is done via `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` ,
`std::asymmetric_coroutine<T>::pull_type::operator()()` for asymmetric coroutines and
`std::symmetric_coroutine<T>::call_type::operator()()` ,
`std::symmetric_coroutine<T>::yield_type::operator()()` for symmetric coroutines.

coroutine-function: The *coroutine-function* returns `void` and takes its counterpart-coroutine as argument, so that using the coroutine passed as argument to *coroutine-function* is the only way (besides simply returning) to transfer data and execution control to another execution context.

For `std::asymmetric_coroutine<T>::pull_type` the *coroutine-function* is entered at `std::asymmetric_coroutine<T>::pull_type` construction. For `std::asymmetric_coroutine<T>::push_type` the *coroutine-function* is not entered at `std::asymmetric_coroutine<T>::push_type` construction but entered by the first invocation of `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` .

For `std::symmetric_coroutine<T>::call_type` the *coroutine-function* is not entered at `std::symmetric_coroutine<T>::call_type` construction but entered by the first invocation of `std::symmetric_coroutine<T>::call_type::operator()()` .
`std::symmetric_coroutine<T>::yield_type` are always synthesized by the framework.
After execution control is returned from *coroutine-function* the state of the coroutine can be checked via `std::asymmetric_coroutine<T>::pull_type::operator bool()` ,
`std::asymmetric_coroutine<T>::push_type::operator bool()` and
`std::symmetric_coroutine<T>::call_type::operator bool()` returning `true` if the coroutine is still valid (*coroutine-function* has not terminated).
Unless `T` is `void` , `std::asymmetric_coroutine<T>::pull_type::operator bool()` returning `true` also implies that a data value is available.

Unless `T` is void, the *coroutine-function* of a `std::symmetric_coroutine<T>::call_type` can assume that (a) upon initial entry and (b) after every
`std::symmetric_coroutine<T>::yield_type::operator()()` call, `std::symmetric_coroutine<T>::yield_type::get()`

has a new value available.
However, if `T` is a move-only type, `std::symmetric_coroutine<T>::yield_type::get()` may only be called once before the next `std::symmetric_coroutine<T>::yield_type::operator()()` call.

passing data from a pull-coroutine to main-context: In order to transfer data from a `std::asymmetric_coroutine<T>::pull_type` to the main-context the framework synthesizes a `std::asymmetric_coroutine<T>::push_type` associated with the `std::asymmetric_coroutine<T>::pull_type` instance in the main-context. The synthesized `std::asymmetric_coroutine<T>::push_type` is passed as argument to *coroutine-function*.
The *coroutine-function* must call this `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` in order to transfer each data value back to the main-context.
In the main-context, the `std::asymmetric_coroutine<T>::pull_type::operator bool()` determines whether the coroutine is still valid and a data value is available or *coroutine-function* has terminated
(`std::asymmetric_coroutine<T>::pull_type` is invalid; no data value available). Access to the transferred data value is given by `std::asymmetric_coroutine<T>::pull_type::get()` .

```
std::asymmetric_coroutine<int>::pull_type source( // constructor enters coro-fn
    [&](std::asymmetric_coroutine<int>::push_type& sink){
        sink(1); // push {1} back to main-context
        sink(1); // push {1} back to main-context
        sink(2); // push {2} back to main-context
        sink(3); // push {3} back to main-context
        sink(5); // push {5} back to main-context
        sink(8); // push {8} back to main-context
    });

while(source){             // test if pull-coroutine is valid
    int ret=source.get(); // access data value
    source();             // context-switch to coroutine-function
}
```

passing data from main-context to a push-coroutine: In order to transfer data to a `std::asymmetric_coroutine<T>::push_type` from the main-context the framework synthesizes a `std::asymmetric_coroutine<T>::pull_type` associated with the `std::asymmetric_coroutine<T>::push_type` instance in the main-context. The synthesized `std::asymmetric_coroutine<T>::pull_type` is passed as argument to *coroutine-function*.
The main-context must call this `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` in order to transfer each data value into the *coroutine-function*.
Access to the transferred data value is given by `std::asymmetric_coroutine<T>::pull_type::get()` .

```
std::asymmetric_coroutine<int>::push_type sink( // constructor does NOT enter coro-fn
    [&](std::asymmetric_coroutine<int>::pull_type& source){
        for (int i:source){
            std::cout << i << " ";
        }
    });

std::vector<int> v{1,1,2,3,5,8,13,21,34,55};
for(int i:v){
    sink(i); // push {i} to coroutine-function
}
```

passing data to a symmetric-coroutine: In order to transfer data to a `std::symmetric_coroutine<T>::call_type` from the main-context the framework synthesizes a `std::symmetric_coroutine<T>::yield_type` associated with the `std::symmetric_coroutine<T>::call_type` instance in the main-context. The synthesized `std::symmetric_coroutine<T>::yield_type` is passed as argument to *coroutine-function*.
The main-context must call this `std::symmetric_coroutine<T>::call_type::operator()()` in order to transfer each data value into the *coroutine-function*.
Access to the transferred data value is given by `std::symmetric_coroutine<T>::yield_type::get()` .

```
// constructor does NOT enter coroutine-function
std::symmetric_coroutine<int>::call_type coro(
    [&](std::symmetric_coroutine<int>::yield_type& yield){
        for (;;) {
            std::cout << yield.get() << "␣";
            yield(); // jump back to starting context
         }
    });

coro(1); // transfer {1} to coroutine-function
coro(2); // transfer {2} to coroutine-function
coro(3); // transfer {3} to coroutine-function
coro(4); // transfer {4} to coroutine-function
coro(5); // transfer {5} to coroutine-function
```

accessing parameters: Parameters returned from or transferred to the *coroutine-function* can be accessed with `std::asymmetric_coroutine<T>::pull_type::get()` or `std::symmetric_coroutine<T>::yield_type::get()` .

Splitting-up the access of parameters from context switch function enables to check if `std::asymmetric_coroutine<T>::pull_type` is valid after return from `std::asymmetric_coroutine<T>::pull_type::operator()()` , e.g. `std::asymmetric_coroutine<T>::pull_type` has values and *coroutine-function* has not terminated.

```
std::asymmetric_coroutine<std::tuple<int,int>>::push_type sink(
    [&](std::asymmetric_coroutine<std::tuple<int,int>>::pull_type& source){
        // access tuple {7,11}; x==7 y==1
        int x,y;
        std::tie(x,y)=source.get();
    });

sink(std::make_tuple(7,11));
```

Parameters passed via `std::symmetric_coroutine<T>::call_type` are accessed via `std::symmetric_coroutine<T>::yield_type::get()` inside the symmetric coroutine.

```
std::symmetric_coroutine<std::tuple<int,int>>::call_type call(
    [&](std::symmetric_coroutine<std::tuple<int,int>>::yield_type& yield){
        // access tuple {7,11}; x==7 y==1
        int x,y;
        std::tie(x,y)=yield.get();
    });

call(std::make_tuple(7,11));
```

exceptions: An exception thrown inside a `std::asymmetric_coroutine<T>::pull_type` 's *coroutine-function* before its first call to `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` will be re-thrown by the `std::asymmetric_coroutine<T>::pull_type` constructor. After a `std::asymmetric_coroutine<T>::pull_type` 's *coroutine-function*'s first call to `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` , any subsequent exception inside that *coroutine-function* will be re-thrown by `std::asymmetric_coroutine<T>::pull_type::operator()()` .

An exception thrown inside a `std::asymmetric_coroutine<T>::push_type` 's *coroutine-function* will be re-thrown by `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` .

An uncaught exception inside a `std::symmetric_coroutine<T>::call_type` 's *coroutine-function* will call `std::terminate()` .

exit a *coroutine-function*: A *coroutine-function* is exited with a simple return statement. Returning from a `std::asymmetric_coroutine<T>::pull_type` 's *coroutine-function* jumps back to the main-context that invoked it. If the *coroutine-function* did not execute `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` , control resumes after the `std::asymmetric_coroutine<T>::pull_type` 's constructor. If the *coroutine-function* has executed `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` , control resumes after the most recent

`std::asymmetric_coroutine<T>::pull_type::operator()()` . From that point on, `std::asymmetric_coroutine<T>::pull_type::operator bool()` will return `false` .
Returning from a `std::asymmetric_coroutine<T>::push_type` 's *coroutine-function* jumps back to the main-context that invoked it. Control resumes after the most recent `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` . From that point on, `std::asymmetric_coroutine<T>::push_type::operator bool()` will return `false` .
Returning from a `std::symmetric_coroutine<T>::call_type` 's *coroutine-function* jumps back to the calling `std::symmetric_coroutine<T>::call_type::operator()()` at the start of symmetric coroutine chain. That is, symmetric coroutines do not have a strong, fixed relationship to the caller as asymmetric coroutines do. From that point on, `std::symmetric_coroutine<T>::call_type::operator bool()` will return `false` .

Other libraries and proposals

Mordor: Mordor[13] is another C++ library implementing cooperative multitasking in order to achieve high I/O performance. The difference from this design is that this proposal focuses on enhanced control flow, while Mordor[13] abstracts on the level of tasking: providing a cooperatively scheduled fiber engine.

AT&T Task Library: Another design of a task library was published by AT&T[14] describing a tasking system with non-preemptive scheduling.
`std::asymmetric_coroutine<T>::push_type` / `std::asymmetric_coroutine<T>::pull_type` does not provide scheduling logic but could be used as the basic mechanism for such a tasking abstraction.

boost.fiber: boost.fiber[9] uses symmetric coroutines in order to support cooperative multi-tasking. The interface is similar to `std::thread` , e.g. synchonization primitives like `mutex` , `condition_variable` and `future` are provided.

C++ proposal: resumable functions (N3328[4]): This proposal is a library superset of N3328: the *resumable function* can be implemented on top of coroutines. The proposed coroutine library does not require memory allocation for the future on a context switch and does not require language changes (no keywords like *resumable* and *await*).
As described in N3328 section 3.2.5 'Function Prolog' - the body of a *resumable function* is transformed into a switch statement. This is similar to the stackless coroutines of Python and C#.
A proof-of-concept how *await* could be built upon boost.coroutine[8] has already been implemented in await_emu[12].

Without stackfulness and first-class semantics, some useful execution control flows cannot be supported (for instance cooperative multitasking, checkpointing) and recursive problems such as the 'same fringe' example become much more difficult.

C++ proposal: Library Foundations for Asynchronous Operations, Revision 1 (N3964[5]): N3964 discusses how boost.asio[6]'s generalized `CompletionToken` support facilitates integrating asynchronous operations with coroutines and fibers. The paper proposes this mechanism for use in any asychronous library functions.

Proposed Wording

std::asymmetric_coroutine<>::pull_type

Defined in header `<coroutine>` .

```
template<class T> class asymmetric_coroutine<T>::pull_type;
```

```
template<class T> class asymmetric_coroutine<T&>::pull_type;
```

```
template<> class asymmetric_coroutine<void>::pull_type;
```

The class `std::asymmetric_coroutine<T>::pull_type` provides a mechanism to receive data values from another execution context.

member types

`iterator`	std::input_iterator (not defined for asymmetric_coroutine<void>::pull_type template specialization)

member functions

(constructor) constructs new coroutine

`pull_type();`	(1)
`pull_type(Function&& fn);`	(2)
`pull_type(pull_type&& other);`	(3)
`pull_type(const pull_type& other)=delete;`	(4)

1) creates a `std::asymmetric_coroutine<T>::pull_type` which does not represent a context of execution

2) creates a `std::asymmetric_coroutine<T>::pull_type` object and associates it with a execution context

3) move constructor, constructs a `std::asymmetric_coroutine<T>::pull_type` object to represent a context of execution that was represented by *other*, after this call *other* no longer represents a coroutine

4) copy constructor is deleted; coroutines are not copyable

Notes

Return values from the *coroutine-function* are accessible via `std::asymmetric_coroutine<T>::pull_type::get()` .
If the *coroutine-function* throws an exception, this exception is re-thrown when the caller returns from `std::asymmetric_coroutine<T>::pull_type::operator()()` .

Parameters

other another coroutine object with which to construct this coroutine object

fn function to execute in the new coroutine

Exceptions

1), 3) noexcept specification: `noexcept`

2) `std::system_error` if the coroutine could not be started - the exception may represent a implementation-specific error condition; re-throw user defined exceptions from *coroutine-function*

Example

```
std::asymmetric_coroutine<int>::pull_type source(
    [&](std::asymmetric_coroutine<int>::push_type& sink){
        int first=1,second=1;
        sink(first);
        sink(second);
        for(int i=0;i<8;++i){
            int third=first+second;
            first=second;
            second=third;
            sink(third);
        }
    });

for(auto i:source)
    std::cout << i <<  "␣";

std::cout << "\nDone" << std::endl;

output:
1 1 2 3 5 8 13 21 34 55
Done
```

(destructor) destroys a coroutine

`~pull_type();`	(1)

1) destroys a `std::asymmetric_coroutine<T>::pull_type` . If that `std::asymmetric_coroutine<T>::pull_type` is associated with a context of execution, then the context of execution is destroyed too. Specifically, its stack is unwound.

operator= moves the coroutine object

`pull_type & operator=(pull_type&& other);`	(1)
`pull_type & operator=(const pull_type& other)=delete;`	(2)

1) assigns the state of *other* to *this using move semantics

2) copy assignment is deleted; coroutines are not copyable

Parameters

other another coroutine object to assign to this coroutine object

Return value

***this**

Exceptions

1) noexcept specification: `noexcept`

operator bool indicates whether context of execution is still valid and a return value can be retrieved, or *coroutine-function* has finished

`operator bool();`	(1)

1) evaluates to true if coroutine is not complete (*coroutine-function* has not terminated)

Exceptions

1) noexcept specification: `noexcept`

operator() jump context of execution

`pull_type & operator()();`	(1)

1) transfer control of execution to *coroutine-function*

Notes

It is important that the coroutine is still valid (`operator bool()` returns `true`) before calling this function, otherwise it results in undefined behaviour.

Return value

***this**

Exceptions

1) `std::system_error` if control of execution could not be transferred to other execution context - the exception may represent a implementation-specific error condition; re-throw user-defined exceptions from *coroutine-function*

get accesses the current value from *coroutine-function*

`R get();`	(1)	(member of generic template)
`R& get();`	(2)	(member of generic template)
`void get()=delete;`	(3)	(only for asymmetric_coroutine<void>::pull_type template specialization)

1) access values returned from *coroutine-function* (if move-only, the value is moved, otherwise copied)

2) access reference returned from *coroutine-function*

Notes

It is important that the coroutine is still valid (`operator bool()` returns `true`) before calling this function, otherwise it results in undefined behaviour.
If type T is move-only, it will be returned using move semantics. With such a type, if you call `get()` a second time before calling `operator()()` , `get()` will throw an exception – see below.

Return value

R return type is defined by coroutine's template argument

void coroutine does not support `get()`

Exceptions

1) Once a particular move-only value has already been retrieved by `get()` , any subsequent `get()` call throws `std::coroutine_error` with an error-code `std::coroutine_errc::no_data` until `operator()()` is called.

swap

swaps two coroutine objects

```
void swap(pull_type& other);    (1)
```

1) exchanges the underlying context of execution of two coroutine objects

Exceptions

1) noexcept specification: `noexcept`

non-member functions

std::swap

Specializes `std::swap` for `std::asymmetric_coroutine<T>::pull_type` and swaps the underlying context of lhs and rhs.

```
void swap(pull_type& lhs,pull_type& rhs);    (1)
```

1) exchanges the underlying context of execution of two coroutine objects by calling `lhs.swap(rhs)` .

Exceptions

1) noexcept specification: `noexcept`

std::begin

Specializes `std::begin` for `std::asymmetric_coroutine<T>::pull_type` .

```
template<class R> asymmetric_coroutine<R>::pull_type::iterator begin(coroutine<R>::pull_type& c);    (1)
```

1) creates and returns a `std::input_iterator`

std::end

Specializes `std::end` for `std::asymmetric_coroutine<T>::pull_type` .

```
template<class R> asymmetric_coroutine<R>::pull_type::iterator end(coroutine<R>::pull_type& c);    (1)
```

1) creates and returns a `std::input_iterator` indicating the termination of the *coroutine-function*

Incrementing the iterator switches the execution context.
When a main-context calls `iterator::operator++()` on an iterator obtained from an explicitly-instantiated `std::asymmetric_coroutine<T>::pull_type` , it must compare the incremented value with the iterator returned by `std::end()` . If they are unequal, the *coroutine-function* has passed a new data value, which can be accessed via `iterator::operator*()` . Otherwise the *coroutine-function* has terminated and the incremented iterator has become invalid.

When a `std::asymmetric_coroutine<T>::push_type` 's *coroutine-function* calls `iterator::operator++()` on an iterator obtained from the `std::asymmetric_coroutine<T>::pull_type` passed by the library, control is transferred back to the main-context. The main-context might never pass another data value. From the *coroutine-function*'s point of view, the `iterator::operator++()` call might never return. If it does return, the main-context has passed a new data value, which can be accessed via `iterator::operator*()` .
A function written to compare the incremented iterator with the iterator returned by `std::end()` can be used in either situation.
If the return-type is move-only the first call to `iterator::operator*()` moves the value. After that, any subsequent call to `iterator::operator*()` throws an exception (`std::coroutine_error`) until `iterator::operator++()` is called.
The iterator is forward-only.

Notes
Because `std::asymmetric_coroutine<T>::pull_type::iterator` is an `InputIterator` , you cannot expect to copy an iterator and increment it independently of the original.

Example

```
int j=10;
std::asymmetric_coroutine<int>::pull_type source(
    [&](std::asymmetric_coroutine<int>::push_type& sink){
        for(int i=0;i<j;++i){
            sink(i);
        }
    });

auto e(std::end(source));
for(auto i(std::begin(source));i!=e;++i){
    std::cout << *i << "␣";
}
```

std::asymmetric_coroutine<>::push_type

Defined in header `<coroutine>` .

`template<class T> class asymmetric_coroutine<T>::push_type;`
`template<class T> class asymmetric_coroutine<T&>::push_type;`
`template<> class asymmetric_coroutine<void>::push_type;`

The class `std::asymmetric_coroutine<T>::push_type` provides a mechanism to send a data value from one execution context to another.

member types

`iterator`	std::output_iterator	(not defined for asymmetric_coroutine<void>::push_type template specialization)

member functions

(constructor) constructs new coroutine

`push_type();`	(1)
`push_type(Function&& fn);`	(2)
`push_type(push_type&& other);`	(3)
`push_type(const push_type& other)=delete;`	(4)

1) creates a `std::asymmetric_coroutine<T>::push_type` which does not represent a context of execution

2) creates a `std::asymmetric_coroutine<T>::push_type` object and associates it with a execution context

3) move constructor, constructs a `std::asymmetric_coroutine<T>::push_type` object to represent a context of execution that was represented by *other*, after this call *other* no longer represents a coroutine

4) copy constructor is deleted; coroutines are not copyable

Parameters

other another coroutine object with which to construct this coroutine object

fn function to execute in the new coroutine

Exceptions

1), 3) noexcept specification: `noexcept`

2) `std::system_error` if the coroutine could not be started - the exception may represent a implementation-specific error condition

Notes

If the *coroutine-function* throws an exception, this exception is re-thrown when the caller returns from `std::asymmetric_coroutine<T>::push_type::operator()(Arg&&)` .

Example

```
std::asymmetric_coroutine<std::tuple<int,int>>::push_type sink(
    [&](std::asymmetric_coroutine<std::tuple<int,int>>::pull_type& source){
        // access tuple {7,11}; x==7 y==1
        int x,y;
        std::tie(x,y)=source.get();
    });

sink(std::make_tuple(7,11));
```

(destructor) destroys a coroutine

`~push_type();`	(1)

1) destroys a `std::asymmetric_coroutine<T>::push_type` . If that `std::asymmetric_coroutine<T>::push_type` is associated with a context of execution, then the context of execution is destroyed too. Specifically, its stack is unwound.

operator= moves the coroutine object

`push_type & operator=(push_type&& other);`	(1)
`push_type & operator=(const push_type& other)=delete;`	(2)

1) assigns the state of *other* to *this using move semantics

2) copy assignment operator is deleted; coroutines are not copyable

Parameters

other another coroutine object to assign to this coroutine object

Return value

***this**

Exceptions

1) noexcept specification: `noexcept`

operator bool

indicates if context of execution is still valid, that is, *coroutine-function* has not finished

```
operator bool();    (1)
```

1) evaluates to true if coroutine is not complete (*coroutine-function* has not terminated)

Exceptions

1) noexcept specification: `noexcept`

operator()

jump context of execution

`push_type & operator()(const Arg& arg);`	(1)	(member of generic template)
`push_type & operator()(Arg&& arg);`	(2)	(member of generic template)
`push_type & operator()(Arg& arg);`	(3)	(member only of asymmetric_coroutine<Arg&>::push_type template specialization)
`push_type & operator()();`	(4)	(member only of asymmetric_coroutine<void>::push_type template specialization)

1),2) If *Arg* is move-only, it will be passed using move semantics. Otherwise it will be copied.

Switches the context of execution, transferring *arg* to *coroutine-function*.

Note

It is important that the coroutine is still valid (`operator bool()` returns `true`) before calling this function, otherwise it results in undefined behaviour.

Parameters

arg argument to pass to the *coroutine-function*

Return value

***this**

Exceptions

1) `std::system_error` if control of execution could not be transferred to other execution context - the exception may represent a implementation-specific error condition; re-throw user-defined exceptions from *coroutine-function*

swap

swaps two coroutine objects

```
void swap(push_type& other);    (1)
```

1) exchanges the underlying context of execution of two coroutine objects

Exceptions

1) noexcept specification: `noexcept`

non-member functions

std::swap Specializes `std::swap` for `std::asymmetric_coroutine<T>::push_type` and swaps the underlying context of lhs and rhs.

```
void swap(push_type& lhs,push_type& rhs);    (1)
```

1) exchanges the underlying context of execution of two coroutine objects by calling `lhs.swap(rhs)` .

Exceptions

1) noexcept specification: `noexcept`

std::begin Specializes `std::begin` for `std::asymmetric_coroutine<T>::push_type` .

```
template<class R> asymmetric_coroutine<R>::push_type::iterator begin(coroutine<R>::push_type& c);    (1)
```

1) creates and returns a `std::output_iterator`

std::end Specializes `std::end` for `std::asymmetric_coroutine<T>::push_type` .

```
template<class R> asymmetric_coroutine<R>::push_type::iterator end(coroutine<R>::push_type& c);    (1)
```

1) creates and returns a `std::output_iterator` indicating the termination of the coroutine

Calling `iterator::operator*(Arg&&)` switches the execution context and transfers the given data value.
`iterator::operator*(Arg&&)` returns if other context has transferred control of execution back.
The iterator is forward-only.

Notes

Because `std::asymmetric_coroutine<T>::push_type::iterator` is an `OutputIterator` , you cannot expect to copy an iterator and increment it independently of the original.

Example

```
std::asymmetric_coroutine<int>::push_type sink(
    [&](std::asymmetric_coroutine<int>::pull_type& source){
        while(source){
            std::cout << source.get() << "␣";
            source();
        }
    });

std::vector<int> v{1,1,2,3,5,8,13,21,34,55};
std::copy(std::begin(v),std::end(v),std::begin(sink));
```

std::symmetric_coroutine<>::call_type

Defined in header `<coroutine>` .

```
template<class T> class symmetric_coroutine<T>::call_type;
```

```
template<class T> class symmetric_coroutine<T&>::call_type;
```

```
template<> class symmetric_coroutine<void>::call_type;
```

The class `std::symmetric_coroutine<T>::call_type` provides a mechanism to send a data value from one execution context to another.

member functions

(constructor) constructs new coroutine

`call_type();`	(1)
`call_type(Function&& fn);`	(2)
`call_type(call_type&& other);`	(3)
`call_type(const call_type& other)=delete;`	(4)

1) creates a `std::symmetric_coroutine<T>::call_type` which does not represent a context of execution

2) creates a `std::symmetric_coroutine<T>::call_type` object and associates it with a execution context

3) move constructor, constructs a `std::symmetric_coroutine<T>::call_type` object to represent a context of execution that was represented by *other*, after this call *other* no longer represents a coroutine

4) copy constructor is deleted; coroutines are not copyable

Notes

If the *coroutine-function* throws an exception and this exception is uncatched, `std::terminate()` is called.

Parameters

other another coroutine object with which to construct this coroutine object

fn function to execute in the new coroutine

Exceptions

1), 3) noexcept specification: `noexcept`

2) `std::system_error` if the coroutine could not be started - the exception may represent a implementation-specific error condition; re-throw user defined exceptions from *coroutine-function*

Example

```
std::symmetric_coroutine<std::tuple<int,int>>::call_type call(
    [&](std::symmetric_coroutine<std::tuple<int,int>>::yield_type& yield){
        // access tuple {7,11}; x==7 y==1
        int x,y;
        std::tie(x,y)=yield.get();
    });

call(std::make_tuple(7,11));
```

(destructor) destroys a coroutine

`~call_type();`	(1)

1) destroys a `std::symmetric_coroutine<T>::call_type`. If that `std::symmetric_coroutine<T>::call_type` is associated with a context of execution, then the context of execution is destroyed too. Specifically, its stack is unwound.

operator= moves the coroutine object

`call_type & operator=(call_type&& other);`	(1)
`call_type & operator=(const call_type& other)=delete;`	(2)

1) assigns the state of *other* to *this using move semantics

2) copy assignment is deleted; coroutines are not copyable

Parameters

other another coroutine object to assign to this coroutine object

Return value

***this**

Exceptions

1) noexcept specification: `noexcept`

operator bool

indicates whether context of execution is still valid or *coroutine-function* has finished

```
operator bool();    (1)
```

1) evaluates to true if coroutine is not complete (*coroutine-function* has not terminated)

Exceptions

1) noexcept specification: `noexcept`

operator()

jump context of execution

`call_type & operator()(const Arg& arg);`	(1)	(member of generic template)
`call_type & operator()(Arg&& arg);`	(2)	(member of generic template)
`call_type & operator()(Arg& arg);`	(3)	(member only of symmetric_coroutine<Arg&>::call_type template specialization)
`call_type & operator()();`	(4)	(member only of symmetric_coroutine<void>::call_type template specialization)

1),2) If *Arg* is move-only, it will be passed using move semantics. Otherwise it will be copied.

Switches the context of execution, transferring *arg* to *coroutine-function*.

Notes

It is important that the coroutine is still valid (`operator bool()` returns `true`) before calling this function, otherwise it results in undefined behaviour.

Return value

***this**

Exceptions

1) noexcept specification: `noexcept`

swap

swaps two coroutine objects

```
void swap(call_type& other);    (1)
```

1) exchanges the underlying context of execution of two coroutine objects

Exceptions

1) noexcept specification: `noexcept`

non-member functions

std::swap Specializes `std::swap` for `std::symmetric_coroutine<T>::call_type` and swaps the underlying context of lhs and rhs.

```
void swap(call_type& lhs,call_type& rhs);    (1)
```

1) exchanges the underlying context of execution of two coroutine objects by calling `lhs.swap(rhs)` .

Exceptions

1) noexcept specification: `noexcept`

std::symmetric_coroutine<>::yield_type

Defined in header `<coroutine>` .

```
template<class T> class symmetric_coroutine<T>::yield_type;
```

```
template<class T> class symmetric_coroutine<T&>::yield_type;
```

```
template<> class symmetric_coroutine<void>::yield_type;
```

The class `std::symmetric_coroutine<T>::yield_type` provides a mechanism to receive data values from another execution context and to transfer the execution control to another coroutine.

member functions

(constructor) constructs new coroutine

```
yield_type();                                (1)
```

```
yield_type(yield_type&& other);              (2)
```

```
yield_type(const yield_type& other)=delete;  (3)
```

1) creates a `std::symmetric_coroutine<T>::yield_type` which does not represent a context of execution

2) move constructor, constructs a `std::symmetric_coroutine<T>::yield_type` object to represent a context of execution that was represented by *other*, after this call *other* no longer represents a coroutine

3) copy constructor is deleted; coroutines are not copyable

Notes

Values to the *coroutine-function* are accessible via `std::symmetric_coroutine<T>::yield_type::get()` .
`std::symmetric_coroutine<T>::yield_type` can be synthesized by the library only.

Parameters

other another coroutine object with which to construct this coroutine object

Exceptions

1) - 3) noexcept specification: `noexcept`

(destructor) destroys a coroutine

```
~yield_type();    (1)
```

1) destroys a `std::symmetric_coroutine<T>::yield_type`

operator= moves the coroutine object

`yield_type & operator=(yield_type&& other);`	(1)
`yield_type & operator=(const yield_type& other)=delete;`	(2)

1) assigns the state of *other* to *this using move semantics

2) copy assignment is deleted; coroutines are not copyable

Parameters

other another coroutine object to assign to this coroutine object

Return value

***this**

Exceptions

1) noexcept specification: `noexcept`

operator bool indicates whether the coroutine is still a valid instance

`operator bool();`	(1)

1) evaluates to true if the instance is a valid coroutine

Exceptions

1) noexcept specification: `noexcept`

operator() jump context of execution

`yield_type & operator()();`	(1)
`template< typename X > yield_type & operator()( symmetric_coroutine< T >::call_type & other, X & x);`	(2)
`template<> yield_type & operator()( symmetric_coroutine< void >::call_type & other);`	(3)

1) transfer control of execution to the starting point, e.g invocation of `std::symmetric_coroutine<T>::call_type::operator()()`

2) transfer control of execution to another symmetric coroutine, parameter `x` is passed as value into other's context

3) transfer control of execution to another symmetric coroutine

Notes

It is important that the coroutine is still valid (`operator bool()` returns `true`) before calling this function, otherwise it results in undefined behaviour.

Return value

***this**

Exceptions

1) `std::system_error` if control of execution could not be transferred to other execution context - the exception may represent a implementation-specific error condition; re-throw user-defined exceptions from *coroutine-function*

Example

```
std::symmetric_coroutine<int>::call_type coro_a(
    [&](std::symmetric_coroutine<int>::yield_type& yield){
            std::cout << yield.get() << "\n";
            yield(); // jump back to starting context
            std::cout << yield.get() << "\n";
            std::cout << "coro_a finished" << std::endl;
    });

coro_a(3); // start coro_a with parameter 3

std::symmetric_coroutine<std::string>::call_type coro_b(
    [&](std::symmetric_coroutine<std::string>::yield_type& yield){
            std::cout << yield.get() << "\n";
            yield( coro_a, 7); // resume coro_a with parameter 7
            std::cout << "coro_b finished" << std::endl;
    });

coro_b("abc"); // start coro_b with parameter "abc"
std::cout << "Done" << std::endl;

output:
3
abc
7
coro_a finished
coro_b finished
Done
```

get accesses the current value passed to *coroutine-function*

`R get();`	(1)	(member of generic template)
`R& get();`	(2)	(member of generic template)
`void get()=delete;`	(3)	(only for symmetric_coroutine<void>::yield_type template specialization)

1) access values passed to *coroutine-function* (if move-only, the value is moved, otherwise copied)

2) access reference passed to *coroutine-function*

Notes

If type T is move-only, it will be returned using move semantics. With such a type, if you call `get()` a second time before calling `operator()()`, `get()` will throw an exception – see below.

Return value

R return type is defined by coroutine's template argument

void coroutine does not support `get()`

Exceptions

1) Once a particular move-only value has already been retrieved by `get()`, any subsequent `get()` call throws `std::coroutine_error` with an error-code `std::coroutine_errc::no_data` until `operator()()` is called.

swap swaps two coroutine objects

`void swap(yield_type& other);`	(1)	

1) exchanges the underlying context of execution of two coroutine objects

Exceptions

1) noexcept specification: `noexcept`

non-member functions

std::swap Specializes `std::swap` for `std::symmetric_coroutine<T>::yield_type` and swaps the underlying context of lhs and rhs.

```
void swap(yield_type& lhs,yield_type& rhs);    (1)
```

1) exchanges the underlying context of execution of two coroutine objects by calling `lhs.swap(rhs)` .

Exceptions

1) noexcept specification: `noexcept`

std::coroutine_errc

Defined in header `<coroutine>` .

```
enum class coroutine_errc { no_data };
```

Enumeration `std::coroutine_errc` defines the error codes reported by `std::asymmetric_coroutine<T>::pull_type` or `std::symmetric_coroutine<T>::yield_type` in `std::coroutine_error` exception object.

member constants Determines error code.

`no_data`	`std::asymmetric_coroutine<T>::pull_type` or `std::symmetric_coroutine<T>::yield_type` has no valid data (maybe moved by prior access)

std::coroutine_error

Defined in header `<coroutine>` .

```
class coroutine_error;
```

The class `std::coroutine_error` defines an exception class that is derived from `std::logic_error` .

member functions

(constructor) constructs new coroutine error object.

```
coroutine_error( std::error_code ec);    (1)
```

1) creates a `std::coroutine_error` error object from an error-code.

Parameters

ec error-code

code Returns the error-code.

```
const std::error_code& code() const;    (1)
```

1) returns the stored error code.

Return value

std::error_code stored error code

Exceptions

1) noexcept specification: `noexcept`

what Returns a error-description.

```
virtual const char* what() const;    (1)
```

1) returns a description of the error.

Return value

char* null-terminated string with error description

Exceptions

1) noexcept specification: `noexcept`

References

[1] Conway, Melvin E.. "Design of a Separable Transition-Diagram Compiler". Commun. ACM, Volume 6 Issue 7, July 1963, Articale No. 7

[2] Knuth, Donald Ervin (1997). "Fundamental Algorithms. The Art of Computer Programming 1", (3rd ed.). Addison-Wesley. Section 1.4.2: Coroutines

[3] Moura, Ana Lúcia De and Ierusalimschy, Roberto. "Revisiting coroutines". ACM Trans. Program. Lang. Syst., Volume 31 Issue 2, February 2009, Article No. 6

[4] N3328: Resumable Functions

[5] N3964: Library Foundations for Asynchronous Operations, Revision 1.

[6] boost.asio

[7] boost.context

[8] boost.coroutine

[9] boost.fiber

[10] Wikipedia - 'Application binary interface'

[11] Wikipedia - 'Calling convention'

[12] await_emu by Evgeny Panasyuk

[13] Mordor high performance I/O library

[14] AT&T Task Library

[15] Split Stacks in GCC

[16] channel9 - 'The Future of C++'

[17] Wikipedia - 'Coroutine'

[18] boost::asio::yield_context

A. *jump*-operation for SYSV ABI on x86_64

The assembler code (from boost.context[7]) shows what the *jump*-operation might look like for SYSV ABI on x86_64.

```
1  .text
2  .globl jump
3  .type jump,@function
4  jump:
5      /* first argument, RDI, points to stack (X) from which we jump */
6      /* second argument, RSI, points to stack (Y) to which we jump */
7      /* third argument, RDX, parameter we want to pass to jumped context */

9      /* save current non-volatile registers to stack X */
10     pushq  %rbp  /* save RBP */
11     pushq  %rbx  /* save RBX */
12     pushq  %r15  /* save R15 */
```

```
13      pushq   %r14   /* save R14 */
14      pushq   %r13   /* save R13 */
15      pushq   %r12   /* save R12 */

17      /* switch stacks, prepare to jump */
18      /* store RSP (pointing to context-data) in RDI */
19      movq   %rsp, (%rdi)
20      /* restore RSP (pointing to context-data) from RSI */
21      movq   %rsi, %rsp

23      /* restore non-volatile registers from stack Y */
24      popq   %r12   /* restrore R12 */
25      popq   %r13   /* restrore R13 */
26      popq   %r14   /* restrore R14 */
27      popq   %r15   /* restrore R15 */
28      popq   %rbx   /* restrore RBX */
29      popq   %rbp   /* restrore RBP */

31      /* restore return-address */
32      popq   %r8

34      /* data passing */
35      movq   %rdx,   %rax   /* use third arg as return value after jump */
36      movq   %rdx,   %rdi   /* use third arg as first arg in context function */

38      /* context switch */
39      jmp    *%r8   /* indirect jump to context via restored return address */
40  .size jump,.-jump
```

Register `rdi` takes the stack-address of to the current execution context X (containing the *control-block*) and register `rsi` contains the stack-address of the target execution context Y (containing the *control-block*) to be resumed.

In lines 10-15 the contents of the current non-volatile registers are pushed on the *stack* of X.

Lines 19 and 21 exchange the stack-pointers - the current stack-pointer is stored in and returned via the first argument (`rdi`). The address of the other context, contained in `rsi` , is assigned to the stack-pointer `rsp` .

The next block (lines 24-29) restores the contents of non-volatile registers for the execution context Y.

Line 32 pops the address of the instruction which should be executed in Y to register `r8` .

Lines 35 and 36 allow to transfer data (as return value in Y) between context jumps.

The next line transfers execution control (*branch-and-link*) to Y by executing the instruction to which `r8` points.